→ Lancer le programme “Spyder” (indiqué Anaconda 3 entre parenthèses) et revenez lire la suite le temps de patienter...

I Environnement de travail

1.1 Les dossiers réseau

Vous disposez d'un espace de travail sur le réseau du lycée. Il s'agit , dans l'explorateur de fichier, d'un disque nommé “Home” qui se situe sur le réseau. Vous devrez enregistrer les fichiers nécessaires au travail en TD d'informatique dans un dossier de ce lecteur, et pas sur la machine locale (ie. dans un sous-dossier de C:).

1.2 Bonnes pratiques

Dès le début d'un TD, créez un dossier dans votre répertoire personnel qui contiendra l'énoncé et éventuellement d'autres documents. “TD3” paraît être un nom convenable pour le TD de ce jour. (Nous devons donc être dans un dossier de la forme K:/Informatique/TD3).

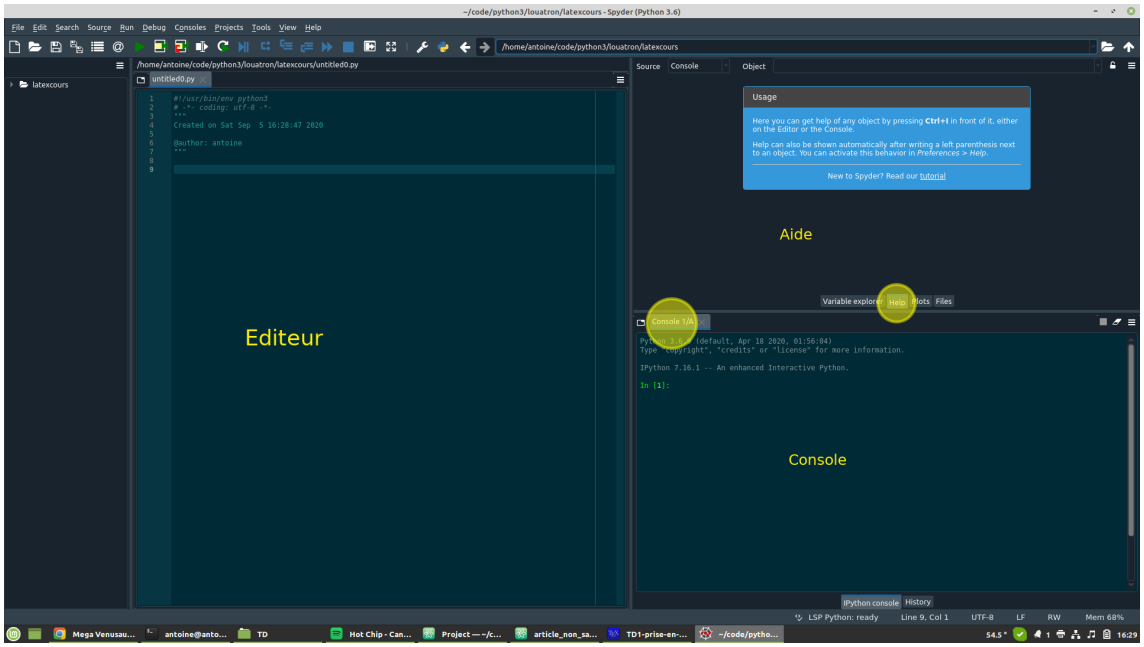
Vous aurez régulièrement des *scripts* python à créer (des fichiers texte ayant l'extension .py). Pensez à sauvegarder très régulièrement, éventuellement en utilisant le classique **CTRL-S** (ou le menu fichier).

1.3 L'IDE

Python, comme tout langage informatique, s'écrit dans un fichier texte. Il faut donc utiliser un éditeur de texte pour créer ou modifier les scripts.
Spyder sera notre éditeur par défaut. Mais ce programme est bien plus qu'un simple éditeur de texte. Vous pouvez voir au moins trois cadres

1. l'éditeur en lui-même, qui nous permettra d'écrire du texte dans un fichier
2. la console qui est le programme python.exe en lui même. Il s'agit d'un “interpréteur” (ou console) : vous tapez une ou plusieurs commandes (en langage Python évidemment) et l'interpréteur calcule le résultat de ces commandes.
3. l'inspecteur d'objet qui nous permettra souvent d'avoir accès à l'aide. Il y a également un onglet Variable explorer qui permet de connaître le contenu des variables actuellement définies dans la console.

La gestion de ces cadres s'effectue via le menu “Affichage”.



L'éditeur sert à écrire du code, mais pas à l'exécuter. La console sert à exécuter du code soit ligne par ligne soit tout un fichier directement (voir plus loin).

Coin Culture Un éditeur de texte qui possède en plus des fonctionnalités pour programmer (comme une console) est appelé un IDE pour Integrated Development Environment.

Lien avec le fonctionnement de Capytale

Vous avez peut-être déjà remarqué le menu dans Capytale permettant de spécifier le type de chaque cellule composant un document. Les deux types que nous utilisons régulièrement sont : *code* et *markdown*

- Les cellules markdown sont des cellules qui ne contiennent que du texte et pas de code.
- les cellule *code* sont en fait des éditeurs de texte (la colonne de gauche dans Spyder) et l'appui sur MAJ + ENTRÉE (ou le clic sur "exécuter") envoie tout votre texte dans la console python qui s'exécute en arrière plan de Capytale.

Si votre Spyder ne fonctionne pas, vous pouvez faire ce TP sur Capytale, à condition d'exécuter les cellules de code où on vous demande d'interagir avec la console

1.4 Aide de python

Il y a plusieurs sources d'aide pour programmer en Python. Dans l'ordre :

1. la documentation de python (menu Aide ou https://docs.python.org/3/)
2. l'aide intégrée à la console (cette commande est à taper dans la console, pour tester son effet) :

```
help(abs)  # montre l'aide de la fonction abs
```

   Durant toute l'année, le contenu des cadres pourra être exécuté directement dans la console, pour observer l'effet. Méfiez-vous des copier-coller en provenance d'un PDF, ils réservent souvent de mauvaises surprises.
   Pour Jupyter notebook (le programme que nous utilisons sur Capytale), un bloc de code bleu de l'énoncé = une cellule de type *code* que l'on exécute.
3. le professeur pendant les TD
4. Google...

II Textes

2.1 Dans les épisodes précédents

1. Que contient la liste `L` après avoir exécuté les lignes suivantes ? Quelle est la longueur (`len(L)`) ? Répondez à ces questions avant de tester

```
L = [1,2,3]
L.append([1,2])
```

2. Même question, en considérant que l'on part de la liste de la question précédente (c'est à dire que vous pouvez continuer vos tests sans redéfinir `L`)

```
L = L + L
```

3. Quelles valeurs prend la variable `i` lors de la boucle suivante ? Que contient la liste `L` une fois la boucle terminée ?

```
for i in range(len(L)):
    L[i] = 12
```

4. Quelles sont les valeurs successives prises par la variable `var` ? Écrire sur une feuille la valeur de la liste `L` après cette boucle.

```
for var in [2, 4, 6]:
    L[i] = L[i] % 2
```

2.2 Manipulations de base

Création de texte

Pour créer des données de texte (que l'on appelle chaînes de caractères, chaque lettre étant un caractère) en python :

```
# Ne pas faire de copier-coller directement dans la console.
t1 = 'Ceci est un texte, on remarque la présence de guillemets.'
t2 = '123456'  # ceci aussi est un texte
```

Pour délimiter un texte, on peut utiliser au choix des guillemets simples comme dans l'exemple précédent (touche 4 du clavier) ou des guillemets doubles (touche 3).

Opérations possibles

Exercice 1
Tester les commande suivantes dans la console, en ayant d'abord créé deux variables t1 et t2 contenant du texte (pourquoi pas les exemples précédents)

1. Premièrement

```
t1[0],t1[1]
```

   De manière générale, quel est l'effet de `t1[i]` ? Quelles sont les valeurs de `i` qui ne conduisent pas à une erreur ?

2. Ensuite

```
len(t1)
```

   Que retournerai la commande `len(t2)` ? Tester.

3. Effet du +

```
t1 + t2, t1 + '3'
```

   Expliquer rapidement l'effet de l'addition de deux chaînes en python. Trouver dans le TP précédent une opération similaire.

4. Multiplication par un entier

```
'a'*7
```

   Quel est l'effet de la multiplication d'une chaîne par un entier ? Peut-on effectuer cette opération sur une liste ?

Boucle for sur les chaînes

Les chaînes sont des itérables au même titre que les listes et les intervalles. Ainsi si on dispose d'une chaîne `s` on a deux méthode pour accéder à chaque caractère de `s` (ces exemples de code ne sont pas exécutables dans la console) :

```
for car in s:
    # car va prendre comme valeurs successives les caractères de s
    # dans l'ordre
```

ou, si on veut en plus accéder aux positions (nommées indices, comme pour les listes) des caractères

```
for i in range(len(s)):
    # comme pour les listes, l'intervalle contenant les indices des éléments de s
    # est donné par range(len(s))
    # Dans ce cas c'est s[i] qui contient les caractères successifs de s
```

2.3 Premiers algorithmes

Avant d'aborder les exercices suivant, ouvrez dans l'éditeur le fichier **tp3.py**. Ce fichier contiendra notre code, mais comme dans Jupyter notebook il faudra exécuter ce code .

Pour exécuter tout le code contenu dans un fichier, on utilise au choix la flèche verte situé juste au dessus du cadre de l'éditeur ou la touche F5. Cette procédure à l'avantage de sauvegarder sur le disque dur le fichier exécuté.

Si votre Spyder ne fonctionne pas et que vous utilisez Jupyter notebook, ouvrez le fichier **tp3.py** grâce au clic droit>ouvrir avec notepad++ et copier les fonctions dans Jupyter au fur et à mesure de vos besoins. Une fonction = une cellule pour le code + une cellule pour tester cette fonction

Exercice 2
Compléter la fonction `occurrences(txt, c)` qui retourne le nombre de fois où le caractère `c` apparaît dans la chaîne `txt`.
Vous pouvez vous inspirer librement des documents précédents (TP et activités).
Un exemple de test possible pour votre fonction :

```
IN [1] : runfile(......)  # c'est la ligne qui apparaît quand on exécute tout le fichier
IN [2] : occurrences('Ceci est un texte.', 'e')  # on exécute une commande utilisant notre fonction
OUT [2] : 4
```

qui signifie que la lettre e apparaît 4 fois dans la chaîne fournie.

Une précision importante sur les fonctions et sur le mot clé `return` : si on exécute un `return` dans une fonction, la fonction s'arrête immédiatement et renvoie le résultat fourni (la valeur se situant après le `return`, sur la même ligne)

Exercice 3
On se propose dans cet exercice de tester si deux mots notés `mot1` et `mot2` sont anagrammes l'un de l'autre c'est à dire sont composés des mêmes lettres, mais pas forcément dans le même ordre. Par exemple "sujet" et "juste" sont anagrammes l'un de l'autre.
Voici un premier algorithme pour répondre à la question :

- Pour chaque caractère `c` de `mot1`, on compte le nombre d'occurrences de `c` dans `mot1` et dans `mot2`.
- Si ces nombres sont différents, on est sûr que la réponse à notre question est non. Sinon on poursuit le parcourt de `mot1`
- Une fois chaque caractère examiné et si tous les nombres coïncident, nos deux mots sont anagrammes l'un de l'autre.

1. Appliquer à la main cet algorithme sur les mots “niche” et “chien” puis sur “niche” et “chiens”. Expliquer pourquoi la réponse de notre algorithme n'est pas correcte dans le deuxième cas. Que pouvons-nous vérifier en plus pour nous assurer que `mot1` et `mot2` sont bien anagrammes l'un de l'autre ?
2. Compléter la fonction `sont_anagrammes1(mot1, mot2)` qui retourne un booléen indiquant si `mot1` et `mot2` sont anagrammes l'un de l'autre. Les booléens en python se notent `True` et `False` (exactement comme ça, en particulier avec une majuscule)

   ```
   type(True)
   ```

3. Tester votre fonctions avec les exemples de la première question, puis avec “aaaabbbb” et “abababab”. Quel reproche peut-on faire à notre algorithme ?

Exercice 4
Compléter la fonction `demi_escalier(n)`. On souhaite par exemple que le résultat de `demi_escalier(4)` soit la chaîne affichée comme suit :

```
#
##
###
####
```

Plusieurs points sont à prendre en compte :

- La valeur de retour de la fonction doit être une unique chaîne de caractère.
- Pour passer à la ligne dans une chaîne de caractère, on utilise le caractère spécial ’\n’. Par exemple, la chaîne retournée par `escalier(2)` est la chaîne

  ```
  '#\n##\n'
  ```

- La représentation en python d'une chaîne (comprendre : en mémoire vive) est différente de son affichage sur l'écran. Pour obtenir un résultat visuel (mais qui n'a aucune signification pour python) on utilise la fonction `print`.
  Ainsi le résultat affiché est obtenu par

  ```
  print(demi_escalier(4))
  ```

Exercice 5
Calculer sur feuille la longueur de la chaîne retournée par `demi_escalier(n)` où $n \geqslant 1$ est un entier.
Vérifier dans la console votre formule sur quelques exemples.

Exercice 6
Compléter la fonction `escalier` de telle manière que le résultat de `print(escalier(3, 5))` soit

```
  #####
 #######
#########
```

Cette fois il faut compter précisément le nombre d'espaces au début de chaque ligne.

Plus difficile

1. Ecrire une fonction qui teste si les caractères d'une chaîne sont 2 à 2 distincts.
2. Ecrire une fonction `est_sous_chaine_pos(txt, ch, k)` qui teste si la chaîne `ch` est une sous-chaîne de `txt` à partir de la position k, ou encore, en notant m la longueur de `ch` :

   $$\forall i \in [0, m-1]\ txt[k+i] == ch[i].$$

   En déduire une fonction qui teste si une chaîne `ch` est une sous-chaîne de `txt`.
3. Créer une fonction `compte_mots(txt)` qui compte le nombre de mots contenus dans le texte donné. Un mot est une suite de caractères différents de l'espace.
4. En utilisant la fonction `chr` (et la fonction `ord`, consulter l'aide dans la console et tester des exemples dans la console), créer la fonction `caractere_majoritaire(txt)` qui retourne le caractère apparaissant le plus de fois dans la chaîne `txt` (le premier trouvé en cas d'égalité). On considère que toutes nos chaînes sont composées de lettres minuscules, sans accent ni caractères spéciaux. On ne comptera pas les espaces. Ainsi les seules lettres à prendre en compte sont les caractères de 'a' à'z'.
5. On considère dans cet exercice des chaînes de caractère contenant une expression arithmétique (composée d'entiers, d'opérations et de parenthèses). Par exemple on pourra utiliser la chaîne "4 * (3 + 5)" pour le premier test.

   On souhaite savoir si les parenthèses sont bien placées et font que l'expression peut avoir du sens. Par exemple "4 * (3 + 5" n'a pas de sens, ni "4 * 3 + 5)"

   Écrire une fonction `parentheses` qui prend comme argument une chaîne et retourne un booléen indiquant si l'expression est bien parenthésée.